INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GAMES : Gestion d'arbres d'états multiples en SMECI

Stéphane LE MÉNEC

N° 158
Juillet 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Games : Gestion d'Arbres d'Etats Multiples en Smeci

Stéphane Le Ménec

Projet SECOIA
INRIA Sophia Antipolis
2004, Route des Lucioles, B.P. 93
06902 Sophia Antipolis Cedex

lemenec@sophia.inria.fr

Games : Gestion d'Arbres d'Etats Multiples en Smeci[1]

Résumé : SMECI est un générateur de systèmes multi-experts au sens multi-expertises. Initialement en SMECI, on peut charger dans un système expert plusieurs bases de connaissances, la connaissance de plusieurs experts. GAMES propose de développer plusieurs systèmes experts avec chacun un arbre de tâches et un arbre d'états, chaque système pouvant alors utiliser des connaissances partagées ou propres.

Games : A tool to expand simultaneously several state trees in Smeci

Abstract : SMECI is an expert system shell which manipulates knowledge from several experts. The standard SMECI allows to load many knowledge bases into a single expert system, with a dedicated knowledge base for each expert. The approach of GAMES is to develop several expert systems within the same SMECI session; each of them with its own task- and state- tree. Each expert system may then use its proper and/or shared knowledge.

[1]Smeci : Système Multi-Experts de Conception en Ingénierie.

Table des matières

1 Introduction **5**

1.1 Le générateur de systèmes experts Smeci 5

1.2 Format et conventions typographiques 6

2 Interface **7**

2.1 Variables globales 7

2.2 Hiérarchie de systèmes experts 7

2.3 Valeurs par défaut 8

2.4 Création de systèmes experts 8

2.5 Fonctions de recherche 10

2.6 Chargement de connaissances 11

2.7 Destruction d'un système expert 14

2.8 Passage d'un système dans un autre 14

2.9 Raisonnement avec des systèmes multiples 15

2.10 Copie d'objets d'un système vers un autre 16

2.11 Panneau S.Managers 16

2.12 Organisation des fichiers de Games 17

3 Exemple **19**

3.1 Un jeu comme exemple 19

3.2 Représentation du jeu en Smeci avec Games 19

3.3 Fichier Initialise.ll 22

3.4 Initialise.rul 24

3.5 Descend.ll 25

3.6 Descend.met 25

3.7 Descend.rul 26

4 Résultats **29**

4.1 Un exemple de grille 29

4.2 Liste des chemins 30

4.3 Quelques arbres d'états 33

5 Conclusion **35**

1 INTRODUCTION

GAMES permet la manipulation dans la même session SMECI de plusieurs systèmes experts. Par systèmes experts, on entend ici plusieurs bases de connaissances, mais également plusieurs arbres d'états.

Ces systèmes experts avec chacun leur base de connaissances (un arbre de catégories, un arbre de tâches, ...) et un arbre d'états différent utilisent le même moteur d'inférences, ils s'exécutent lors de la même session SMECI. On ne développe qu'un système expert à la fois, mais on peut suspendre le travail d'un système pour en activer momentanément un autre.

Comme on n'utilise qu'un seul environnement LE-LISP, tous les systèmes experts ont accès à toutes les fonctions LE-LISP. Les méthodes SMECI étant en réalité des fonctions LE-LISP, les méthodes sont nécessairement partagées.

On permet, dans certains cas particuliers, le partage de catégories SMECI entre plusieurs systèmes experts. Par partage, on entend au moins deux systèmes experts indexant le même objet catégorie. On a écrit également le partage d'objets règles entre plusieurs systèmes. On propose principalement le partage de structures relativement figées. On expliquera les problèmes posés par la modification de connaissances partagées, comme par exemple la réécriture d'une règle partagée.

Un système expert peut copier dans son environnement la connaissance d'un autre système, il peut dupliquer les objets d'un autre environnement à condition que les deux systèmes soient compatibles.

Ce document présente un ensemble de fonctions pour développer des applications multi-experts en SMECI version 1.65-1. Puis, nous donnons à la fin un exemple utilisant les fonctions de GAMES pour manipuler un grand nombre de systèmes experts dans la même session SMECI.

1.1 Le générateur de systèmes experts Smeci

SMECI[1, 2, 3, 4] offre un formalisme de représentation de connaissances sous forme d'objets et de règles regroupées en tâches. Des objets appelés catégories ou classes définissent

une hiérarchie de types, que l'on instancie pour créer les objets modélisant le domaine.

Le raisonnement de SMECI consiste à créer un arbre d'états, encore appelé un arbre de raisonnement obtenu par l'application de règles. Un contrôle complexe, décrit principalement dans les tâches, déclenche de façon concurrente les règles d'une base de règles et développe ainsi les états de l'arbre de raisonnement SMECI en mondes multiples.

Un état référence les objets d'un instant particulier du système, des objets caractérisés par les valeurs de leurs champs.

SMECI permet de gérer un arbre d'états important et possède des fonctions pour le manipuler. On peut entre autre naviguer d'un état à l'autre, définir un ordre de préférence pour développer l'arbre des états, couper des branches et visualiser rapidement les informations de cet arbre.

1.2 Format et conventions typographiques

Le format de description des fonctions décrites dans ce rapport respecte les conventions suivantes.

Le signe :

- $\hookrightarrow$ indique des arguments que l'on fournit,
- $\hookleftarrow$ indique les valeurs que l'on reçoit.

La police :

- `typewriter` est utilisée pour le code.
- sanserif est utilisée pour le type des objets.
- *italic* est utilisée pour le nom des arguments.

2 INTERFACE

2.1 Variables globales

- `$S.Manager$` : objet système expert courant.
- `$liste-SEs$` : liste des objets systèmes experts créés.
- `$smecidir$` : répertoire contenant les fichiers systèmes SMECI `sysobjects.ins` et `systasks.ins`.
- `$editeur-SEs$` : application graphique, éditeur de systèmes experts.

2.2 Hiérarchie de systèmes experts

GAMES propose de créer une hiérarchie de classes sous `S.Manager` pour structurer les différents systèmes experts que l'on va instancier. La catégorie `S.Manager` n'étant pas une véritable catégorie SMECI, on ne peut pas utiliser la fonction SMECI `defScategory` pour créer des sous-classes. On ne crée donc que des classes MICRO-CEYX . Dans l'exemple du chapitre 3, on utilise la classe `Deplace-pion` que l'utilisateur déclare avec la fonction LE-LISP `deftclass`.

```
(deftclass {S.Manager}:Deplace-pion)
```

(send 'GAMES-classe *se*) [*méthode*]

Classe d'un système expert.

↪ *se* objet système expert
↩ symbole

2.3 Valeurs par défaut

(send 'GAMES-nom-defaut *se*) [*méthode*]

Nom par défaut d'un système expert, le nom de sa classe.

↪ *se* objet système expert
↩ symbole

(send 'GAMES-bc-defaut *se*) [*méthode*]

Base de connaissance par défaut d'une classe de systèmes experts.

↪ *se* objet système expert
↩ symbole ou chaîne de caractères

Le premier système expert de Smeci n'a pas de base de connaissances chargée par défaut. C'est pourquoi `GAMES-bc-defaut` renvoie () pour les systèmes experts du type `S.Manager`. Cette méthode permet de charger une base de connaissances pour un système en fonction de son type.

2.4 Création de systèmes experts

(GAMES-creer-se *classe nom-se . bc*) [*fonction*]

Crée et initialise un système expert instance de S.Manager ou d'une sous classe de S.Manager.

↪ *classe* symbole
nom-se symbole
bc symbole ou chaîne de caractères
↩ objet système expert

Note *Dans tout ce rapport par objet système expert, il faut comprendre un objet de type S.Manager ou de classe Micro-Ceyx héritant de S.Manager.*

`GAMES-creer-se` accepte comme valeur de l'argument optionnel *bc* : un symbole, ou une chaîne de caractères et retrouve la base de connaissances à associer au nouveau système expert avec la fonction SMECI (`S-find-kb bc`). Si cet argument n'est pas spécifié, le nouveau système expert aura la base de connaisances par défaut de sa classe ou rien s'il n'y a pas de base de connaissances par défaut définie.

Structure de la fonction GAMES-creer-se :

```
(de GAMES-creer-se (classe nom-se .bc)
        vieux-se : se de depart

        creation d'un objet de type classe

        si nom-se <> ()
           alors nom du nouveau-se = nom-se
        sinon nom du nouveau-se = nom par defaut

        nouveau-se dans $S.Manager$

        demon GAMES-av-cree de nouveau-se

        si bc existe
           alors chargement de bc
        sinon si classe a une bc par defaut
                 alors chargement de bc par defaut

        demon GAMES-si-cree de nouveau-se

        demon GAMES-retour de vieux-se

        retourne nouveau-se)
```

(send 'GAMES-av-cree *se*) [*méthode*]

Démon du nouveau système expert déclenché avant sa création.

↪ *se* objet système expert
↩ ()

On marque () comme valeur renvoyée simplement pour indiquer que l'on n'utilise pas la valeur retournée par cet envoi de message. Comme exemple, on donne la méthode GAMES-av-cree de S.Manager:

```
(de {S.Manager}:av-cree (se)
   (:mes "** Je vais creer un S.Manager ~A" (send 'GAMES-classe se)))
```

(send 'GAMES-si-cree *se*) [*méthode*]

Démon du nouveau système expert déclenché après sa création.

↪ *se* objet système expert
↩ ()

A l'initialisation, le démon si-cree de la classe `S.Manager` ne fait rien.

(send 'GAMES-retour *vieux-se nouveau-se*) [*méthode*]

Message envoyé au système expert de départ après la création d'un nouveau système expert.

↪ *vieux-se* objet système expert
nouveau-se objet système expert
↩ ()

Après création d'un nouveau système, le système expert courant est *nouveau-se*. C'est la méthode `GAMES-retour` qui dit quoi faire, notamment dans quel système continuer après l'initialisation du nouveau système. Si l'utilisateur ne programme pas de nouvelles méthodes, le comportement par défaut est le suivant :

```
;;; {S.Manager}:GAMES-retour (vieux-se nouveau-se) est defini pour
;;; retourner dans vieux-se apres creation de nouveau-se
(de {S.Manager}:GAMES-retour (vieux-se nouveau-se)
    (setq $S.Manager$ vieux-se)
    (:mes "** Je reviens dans le ~A" $S.Manager$)
    ;;; mise a jour de l'editeur de systemes experts
    (if (boundp '$editeur-SEs$)
        (GAMES-raff-editeur $editeur-SEs$)))
```

2.5 Fonctions de recherche

(GAMES-trouver-se *nom*) [*fonction*]

Cherche un système expert de nom indiqué, si on n'en trouve pas, alors cherche un système de classe nom.

↪ *nom* symbole
↩ objet système expert

(GAMES-trouver-se-classe *classe . nom*) [*fonction*]

Cherche un système de classe indiquée et de nom spécifié si le paramètre nom est donné.

↪ *classe* symbole
nom symbole
↩ objet système expert

(GAMES-trouver-tous-ses *classe*) [*fonction*]

Donne tous les systèmes de la classe indiquée, ou tous les systèmes si le paramètre classe vaut ().

↪ *classe* symbole
↩ liste d'objets système expert

2.6 Chargement de connaissances

On définit tout d'abord deux fonctions pratiques lors du chargement de connaissances partagées. Elle permettent de savoir si un fichier a déjà été chargé dans un autre système expert. Le rechargement d'un fichier dans un nouveau système signifie partage des données de ce fichier. Si on veut par exemple recharger une catégorie sans vouloir la partager, il faut sauver la définition de cette classe dans deux fichiers différents.

(GAMES-S-find-collector *nom-fic*) [*fonction*]

Cherche un objet collecteur parmi les collecteurs de tous les systèmes experts créés sauf le système courant.

↪ *nom-fic* symbole ou chaîne de caractères
↩ objet collecteur, structure système S.Collector

Retourne () s'il n'y a pas de système autre que le système courant contenant un collecteur de nom *nom-fic*. SMECI associe aux fichiers de catégories et d'objets des collecteurs du même nom. Lors de la recherche de catégories définies dans un fichier d'un système différent du système courant, il est pratique de manipulèr les objets collecteurs associés aux fichiers.

(GAMES-trouver-se-regles *nom-fic*) [*fonction*]

Cherche si un fichier de règles a déjà été chargé dans un autre système que le système courant.

↪ *nom-fic* symbole ou chaîne de caractères
↩ objet système expert

Retourne () s'il n'y a pas de système avec un fichier de règles de nom *nom-fic*. Cette fonction utilise le fait qu'un système expert rassemble dans une liste tous les noms de fichiers de règles de son environnement.

Les fonctions de ce paragraphe chargent la connaissance de un ou plusieurs fichiers SMECI dans le système expert courant.

(GAMES-charger-bc *bc*) [*fonction*]

Chargement d'une base de connaissances.

↪ *bc* base de connaissances, structure système du type S.KnowledgeBase
↩ ()

Cette fonction est identique à la fonction SMECI `S-load-kb`, si ce n'est qu'elle charge aussi des catégories et des règles partagées entre plusieurs systèmes.

(GAMES-charger-categs *fich*) [*fonction*]

Chargement d'un fichier de catégories SMECI.

↪ *fich* symbole ou chaîne de caractères
↩ ()

Si *fic* a déjà été chargé dans un autre système expert, alors les classes de *fic* sont partagées entre ces deux systèmes, sinon les catégories sont chargées normalement. Quand on décide de partager une classe, cela implique également le partage de ses sous-classes, ce que GAMES ne vérifie pas.

Un objet système expert accède à ses classes grâce à une table de hachage encore appelée tableau. La hiérarchie de catégories d'un système expert est modélisée par de doubles liens entre les classes. Chaque classe possède un pointeur vers sa classe mère et une liste de pointeurs vers ses classes filles. Le pointeur d'une classe vers sa classe mère sert peu à SMECI, principalement seul l'éditeur graphique des catégories utilise cette information. Par contre, la liste des pointeurs vers les filles d'une classe doit être exacte, ce qui impose le partage de toutes les filles d'une catégorie partagée.

Le chargement d'une hiérarchie de catégories partagées sous une classe mère d'un nouveau système nécessite la mise à jour du tableau de ce système et la modification de la liste des filles de la classe mère. Le champ classe mère de la racine de la hiérarchie de classes partagées continue de pointer vers sa classe mère dans le premier système. La structure des catégories SMECI ne permet pas de noter plusieurs classes mères pour une classe partagée. Cela explique pourquoi l'éditeur graphique de catégories de SMECI ne fonctionne que pour une exploration descendante d'une hiérarchie de classes partagées.

(GAMES-charger-regles *fich*) [*fonction*]

Chargement d'un fichier de règles.

↪ *fich* symbole ou chaîne de caractères
↩ ()

Cette fonction charge les règles de *fich* dans le système expert courant comme la fonction SMECI standard `S-load-rule-file`, sauf si ce fichier a déjà été chargé dans un autre système; au quel cas GAMES retrouve les objets règles correspondants qui seront partagés.

Il ne faut pas modifier de règles partagées déjà chargées. En effet, si on modifie une telle règle dans un fichier de règles, on ne peut pas la recharger avec `GAMES-charger-regles` qui réutilise nécéssairement les objets règles du même nom déjà chargés. Et si on applique la fonction `S-load-rule-file` alors les règles chargées ne sont plus partagées.

Pour modifier une connaissance partagée, il faut directement transformer les objets partagés et pas les fichiers définissant ces connaissances partagées. On peut agir de la sorte avec des catégories partagées, mais pas avec des objets règles.

Pour modifier une règle partagée, il faut réécrire cette règle, la compiler dans un des environnements correspondants et recharger le fichier de cette règle avec la fonction SMECI `S-load-rule-file`, puis retourner dans tous les systèmes qui partagent cette règle pour recharger le fichier modifié avec `GAMES-charger-regles`.

(GAMES-charger-methodes *fich*) [*fonction*]

Chargement d'un fichier de méthodes.

↪ *fich* symbole ou chaîne de caractères
↩ ()

Même si les fichiers de méthodes sont des fichiers LE-LISP, l'utilisateur doit tout de même recharger avec `GAMES-charger-methode` les méthodes d'une classe partagée. De façon transparente pour l'utilisateur, GAMES regarde si ce fichier de méthodes a déjà été chargé dans un précédent système et ne le charge que s'il n'a pas déjà été chargé. Mais dans tous les cas, GAMES doit mettre à jour la liste des fichiers de méthodes indexés dans l'objet système expert, ce qui explique pourquoi il est impératif de bien recharger les méthodes de catégories partagées. Lors du chargement pour la première fois d'un fichier de méthodes, `GAMES-charger-methodes` a le comportement de la fonction SMECI standard `S-load-file`.

La modification de méthodes partagées pose un problème comparable à la modification de règles partagées. En effet, si on utilise la fonction `GAMES-charger-methodes`, alors GAMES ne recharge pas le fichier de méthodes en imaginant que l'on fait du partage et pas de la redéfinition. Il faut utiliser dans ce cas la fonction SMECI `S-load-file`. Comme il s'agit de fonctions LE-LISP, tous les systèmes voient alors la modification.

(GAMES-charger-objets *fich*) [*fonction*]

Chargement d'un fichier d'objets.

↪ *fich* symbole ou chaîne de caractères
↩ ()

Cette fonction est équivalente à la fonction SMECI `S-load-object-file`. Les objets SMECI ne sont connus que dans le système où ils sont chargés. Une classe ne connaît pas directement ses instances. Ce sont les objets systèmes experts qui associent les objets à leur classe. Une catégorie partagée entre deux systèmes a des instances dans le premier sytème et d'autres instances dans le second système. GAMES ne partage que des structures entre plusieurs systèmes experts.

(GAMES-charger-ficlisp *fich*) [*fonction*]

Chargement d'un fichier Lisp.

↪ *fich* symbole ou chaîne de caractères
↩ ()

Cette fonction utilise la fonction SMECI `S-load-file`. Tous les systèmes s'exécutent dans la même session LE-LISP, toutes les fonctions sont accessibles depuis tous les systèmes. Un fichier LE-LISP n'a besoin d'être chargé qu'une fois.

2.7 Destruction d'un système expert

(GAMES-detruire *se*) [*fonction*]

Efface un système expert d'une session Smeci.

↪ *se* objet système expert
↩ ()

Cette fonction ne doit pas être appelée sur le système expert courant, elle détruit tous les objets indexés dans l'environnement défini par *se* et retire *se* de la liste `$liste-SEs$`.

Les fonctions de l'interface fonctionnelle d'initialisation d'une session SMECI comme `S-reset-root` permettent de réinitialiser le système expert courant, quel que soit son type sans modifier les autres objets système expert utilisés.

2.8 Passage d'un système dans un autre

(send 'GAMES-va *se*) [*méthode*]

Déclare se comme système courant.

↪ *se* objet système expert
↩ objet système expert

Le changement de système consiste à changer l'objet système expert de la variable globale `$S.Manager$` et à mettre à jour l'interface graphique donnant le système courant.

(GAMES-va-dans-se *se*) [*fonction*]

Déclare se comme système courant.

↪ *se* symbole
↩ objet système expert

On change aussi de système en appelant cette fonction qui utilise la méthode précédente après avoir retrouvé l'objet système de nom *se*

(GAMES-dans-se *se* . *expr*) [*macro*]

Evalue expr en se plaçant momentanément dans le système se et revient ensuite dans l'ancien système.

↪ *se* objet système expert
expr expression Lisp à évaluer
↩ objet système expert

2.9 Raisonnement avec des systèmes multiples

(send 'GAMES-lancer *se* . *tache*) [*méthode*]

Lance se avec comme tâche de départ celle indiquée.

↪ *se* objet système expert
tache symbole
↩ ()

On lance depuis un premier système le raisonnement d'un second système. `GAMES-lancer` peut figurer dans une fonction LE-LISPou dans une règle SMECI. A son appel, on change le système courant et on empile sur la pile de tâches la tâche de nom spécifié si elle existe.

Structure de la méthode `GAMES-lancer` :

```
(de {S.Manager}:GAMES-lancer (nouveau-se . tache)
                {S.Manager}:GAMES-lancer execute depuis vieux-se
                va dans nouveau-se
                demon GAMES-av-lancer de nouveau-se
                (S-push-task ''objet de nom tache'')
                (S-start)
                demon GAMES-ap-lancer de nouveau-se
                retour dans vieux-se)
```

(send 'GAMES-av-lancer *se*) [*méthode*]

Démon déclenché dans le nouveau système avant de lancer son raisonnement.

↪ *se* objet système expert
↩ ()

Exemple de démon `GAMES-av-lancer` :

```
(de {S.Manager}:GAMES-av-lancer (se)
    ;;; mise a jour de l'interface graphique
    (if (boundp '$editeur-SEs$)
        (GAMES-raff-editeur $editeur-SEs$))
    ;;; necessaire en version 1.65 pour que le controle n'oublie pas
    ;;; d'instanciations
    (setq #:smeci:*global-cut* nil))
```

(send 'GAMES-ap-lancer *se*) [*méthode*]

Démon déclenché dans le nouveau système à la fin de son raisonnement avant de retourner dans le système de départ.

↪ *se* objet système expert
↩ ()

Le démon `GAMES-ap-lancer` de la classe `S.Manager` ne fait actuellement rien, on pourrait imaginer qu'il édite pour vérification l'agenda des tâches de *se*.

2.10 Copie d'objets d'un système vers un autre

(GAMES-copier-objet *se1 se2 obj*) [*méthode*]

Le système expert se1 copie chez lui un objet calculé par un autre système expert se2 dont la connaissance est supposée compatible avec la sienne.

↪ *se1* objet système expert
se1 objet système expert
obj objet
↩ ()

GAMES ne vérifie pas la compatibilité des deux systèmes experts. On ne copie que les champs utilisateurs. Le nouvel objet aura le même nom que l'objet copié. Les sous objets sont copiés (copie-récursive), sauf ceux des champs systèmes comme activités. Cette fonction ne permet pas la copie d'objets circulaires.

2.11 Panneau S.Managers

Une fenêtre graphique intitulée `S.Managers` (voir figure 2.1) permet de visualiser le système courant, son type et d'exécuter à l'aide de boutons poussoirs les principales fonctions de GAMES. On peut ainsi en sélectionnant un système dans le sélectionneur de chaînes de caractères et en utilisant le bouton `Va dans` changer de système courant. Les boutons exécutent les fonctions définies précédemment sur le système sélectionné à la souris au préalable. Excepté la fonction `Tout detruire` qui détruit tous les systèmes experts sauf le système sélectionné. Cette fonction s'avère pratique lorsque l'on a beaucoup de systèmes comme dans l'exemple du chapitre 3.

(GAMES-cree-editeur *x y*) [*fonction*]

Crée et affiche un éditeur de systèmes experts en (x,y).

↪ *x* entier
y entier
↩ objet

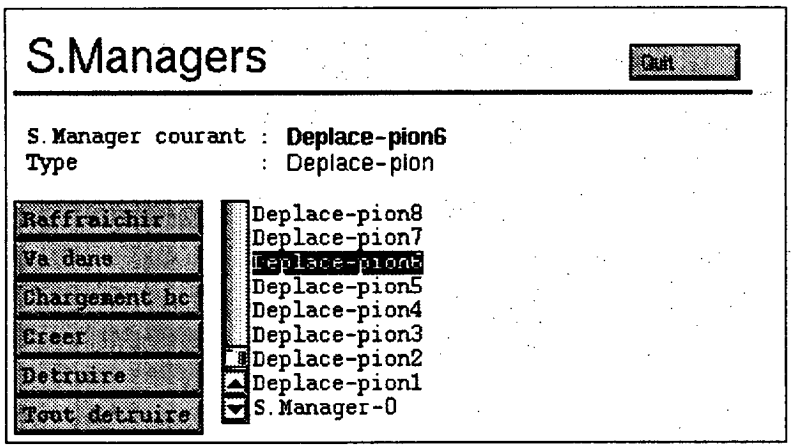

Figure 2.1 : Panneau S.Managers après l'exemple du chapitre 3

(GAMES-raff-editeur *edit*) [*fonction*]

Remet à jour l'éditeur de systèmes experts indiqué.

↪ *edit* objet
↩ objet

Toutes les fenêtres de l'interface graphique de SMECI sont utilisables sur le système expert courant, quelque soit sa classe. Le panneau `S.Managers`, qui utilise la même présentation que les fenêtres SMECI complète l'interface de développement en mode multi experts. On propose donc de rajouter cette petite fenêtre dans le gestionnaire de panneaux par la fonction SMECI `S-add-panel-button` par exemple.

2.12 Organisation des fichiers de Games

GAMES est constitué des quatres fichiers LE-LISP suivants :

- `multi-ex.met` : types des systèmes experts et définition de la macro `GAMES-dans-se`. On écrit dans ce fichier le code que SMECI doit charger avant les fichiers de règles et les fichiers LE-LISP. Ainsi SMECI définit les classe MICRO-CEYX avant de charger les règles qui risquent d'en parler. De même, on définit les macros avant les fonctions qui les utilisent.
- `multi-ex.ll` : fichier des fonctions et des méthodes de GAMES.
- `editeur-multi-ex.ll` : code de la fenêtre graphique `S.Manager`.
- `headband.ll` : définition du bandeau de la fenêtre `S.Manager`.

Seul le fichier `multi-ex.met` où figure la définition des classes de systèmes experts dépend de l'application considérée.

3 EXEMPLE

3.1 Un jeu comme exemple

On définit une grille composée de cases valides marquées d'un 0 et de cases non valides à 1. On place un pion en haut de la grille sur la première ligne composée uniquement de 0. On veut construire un arbre de choix donnant tous les chemins possibles permettant au pion d'arriver sur la dernière ligne de la grille. Cela sachant que le pion descend d'une ligne à chaque fois et qu'il n'a que trois choix possibles (-1, 0 et 1). 0 signifie descendre sur la même colonne alors que -1 correspond à un pion diminuant de 1 son numéro de colonne. En jouant 1, le pion descend d'une ligne en passant sur la colonne suivante.

L'arbre d'états SMECI permet de construire facilement l'arbre de choix de ce jeu. A un nœud de décisions, SMECI crée une branche pour chaque alternative du pion. Les feuilles de l'arbre d'état SMECI donnent alors tous les chemins du pion possibles.

Seulement, on impose une condition supplémentaire. On ne veut pas développer des arbres d'états de profondeur supérieure à `prof-max`. On va donc découper l'arbre de choix initial que l'on se proposait de construire en plusieurs petits arbres d'états de plusieurs systèmes experts.

Le pion arrive toujours à descendre dans la grille sauf si les trois cases sous lui (case de gauche, du centre et de droite par rapport à lui sur la ligne en dessous de la sienne) sont non valides.

3.2 Représentation du jeu en Smeci avec Games

On utilise deux classes de systèmes experts, la classe `S.Manager` et la classe `Deplace-pion`. Le système expert initial de SMECI de type `S.Manager` sert à l'initialisation du problème, puis on crée un système `Deplace-pion` pour chaque sous-arbre à développer.

On crée deux catégories utilisateur `Pion` et `Objet-systeme` dans le fichier `Descend.cat` que l'on partage entre tous les systèmes :

```
Classe : Pion

   champs : l-pos  (liste de couples ligne colonne)
            l-mouv (liste de mouvements)

Classe : Objet-systeme

   champs : grille
            dim-l        (nombre de lignes de la grille)
            dim-c        (nombre de colonnes de la grille)
            profondeur   (profondeur d'un etat d'un sous arbre)
            prof-max     (profondeur maximum d'un sous arbre)
            compteur-ses (pour numeroter les systemes crees)

   (avec le champ compteur-ses declare insensible au contexte)
```

On définit les bases de connaissances du système d'initialisation et des systèmes du type `Deplace-pion`.

```
Initialise-KB : Descend.cat
                Initialise.rul
                multi-ex.met
                Descend.met
                systasks-initialise.ins
                sysobjects-initialise.ins
                $smecidir$/systasks.ins
                multi-ex.ll
                headband.ll
                editeur-multi-ex.ll
                Initialise.ll

Descend-KB    : Descend.cat
                Descend.rul
                Descend.met
                $smecidir$/systasks.ins
                systasks-descend.ins
                sysobjects-descend.ins
                Descend.ll
```

Il faut déclarer dans chaque système le fichier $smecidir$/systasks.ins qui contient les tâches systèmes comme `pop-task`, `push-task`, ...

Les fichiers d'objets `sysobjects-initialise.ins` et `sysobjects-descend.ins` contiennent les objets stratégie et agenda de ces différents systèmes experts. On utilise comme stratégie la recherche en profondeur proposée par SMECI.

Le fichier d'objets `systasks-initialise.ins` définit l'arbre de tâches du système d'initialisation (voir la figure 3.1). L'arbre de tâches utilisé par les systèmes experts du type `Deplace-pion` est donné par le fichier `systasks-descend.ins` (voir la figure 3.2).

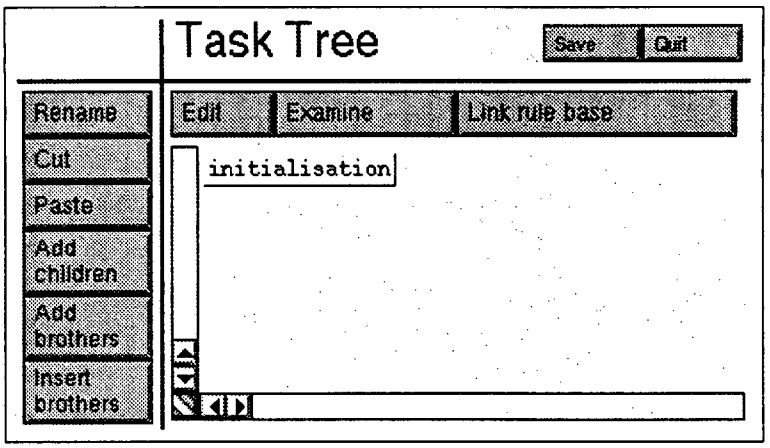

Figure 3.1 : Arbre de tâches du système d'initialisation

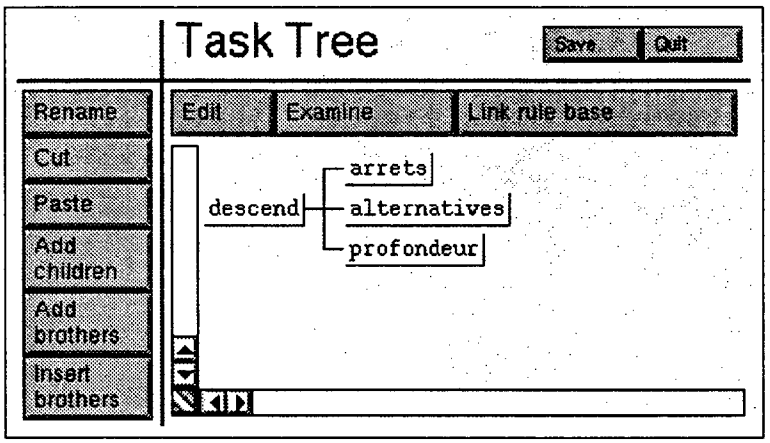

Figure 3.2 : Arbre de tâches des systèmes de classe `Deplace-pion`

La tâche `initialisation` du système d'initialisation utilise la base de règles `initialisation-br` qui applique la règle `initialisation` en mode `instantiate = first-once`.
Les bases de règles des systèmes `Deplace-pion` sont initialisées de la façon suivante :

```
arret-br        : rule-list    : (solution
                                  prof-max)
                  instantiate  : first-once

alternatives-br : rule-list    : (alternatives)
                  instantiate  : first-all
                  firing-mode  : parallel

profondeur-br   : rule-list    : (profondeur)
                  instantiate  : first-once
```

Dans les sections suivantes, nous donnons les quelques règles et fonctions de cet exemple.

3.3 Fichier Initialise.ll

```
(de creer-grille (dim-l dim-c proba-validite)
;;; cree et initialise une grille [(0..(nb-l - 1)) (0..(nb-c - 1))]
    (let ((grille (makearray dim-l dim-c 0)))
         (for (i 1 1 (1- dim-l))
              (for (j 0 1 (1- dim-c))
                   (aset grille i j
                         ;;; probabilite d'avoir un 1 dans
                         ;;; la grille : 1/proba-validite
                         (if (< (random 0 proba-validite) 1) 1 0))))
         grille))

(de affiche-grille (grille dim-l dim-c)
;;; dessine une grille
    (prin "   ")
    (for (j 0 1 (1- dim-c)) (prin j) (prin " "))
    (print)
    (prin "  -")
    (for (j 0 1 (1- dim-c)) (prin "--"))
    (print)
    (for (i 0 1 (1- dim-l))
         (prin i)
         (prin " |")
         (for (j 0 1 (1- dim-c))
              (prin (aref grille i j))
              (prin "|"))
         (print)
         (prin "  -")
         (for (j 0 1 (1- dim-c)) (prin "--"))
         (print)))
```

```
(de dessine ()
;;; dessine la grille donnee par l'objet systeme
    (let ((obj-sys (car (S-all-objects 'Objet-systeme))))
         (affiche-grille grille^obj-sys dim-l^obj-sys dim-c^obj-sys)))

(de dessine-fic (fic)
;;; dessine la grille donnee par l'objet systeme dans fic
    (let ((chan (opena fic)))
         (outchan chan)
         (dessine)
         (close)))

(de ecrit (message fichier)
;;; ecriture de message dans fichier
;;; toutes les ecritures avec outchan se font en ajout a la fin de
;;; fichier
    (let ((chan (opena fichier)))
         (outchan chan)
         (print message)
         (close)))

;;; agenda, strategie et granularite du se d'initialisation
(S-set-agenda self^agenda-initialise)
(S-set-strategy self^strategy-initialise)
;;;(S-state-granularity 'branching)
(S-state-granularity 'rule)

;;; singularite pour un pion bloque au milieu de la grille
(S-define-singularity 'Bloque "Le pion est bloque")
;;; singularite pour un etat a profondeur maximale dans un se
(S-define-singularity 'Prof-max "On continue dans un autre se")

;;; *** ACTIVATION DE LA FENETRE S.MANAGERS ***

;;; editeur multi-experts mis dans la variable globale $editeur-SEs$
;;; par GAMES-cree-editeur
(ifn (boundp '$editeur-SEs$)
     (progn (setq $editeur-SEs$ (GAMES-cree-editeur 10 10))
            (S-add-panel-button "S.Managers" $editeur-SEs$)))

;;; CREATION DE CONFIGURATIONS SMECI

(ifn (boundp 'Config-Trajedit)
     (progn (setq Config-Trajedit t)
            (S-add-config-button "Config-Trajedit"
                    $editeur-SEs$ (S-terminal) (S-S.Manager-panel))))
```

3.4 Initialise.rul

```
defrule initialisation
author steph
explanation
   cree une grille,
   initialise les cases valides de la grille,
   cree un pion,
   place ce pion sur sa position initiale,
   cree un premier se du type Deplace-pion,
   copie l'objet pion dans ce nouveau se,
   et lance ce nouveau se
end-explanation
if $(length $liste-SEs$) = 1
then
create *pion un Pion
       et *obj-sys un Objet-systeme
puis
;;; PARAMETRES ET CONDITIONS INITIALES
variables
   ;;; nombre de lignes et de colonnes de la grille
   ;;; tableau de [(0..(nb-l - 1)) (0..(nb-c - 1))]
   declare nb-l valant 10
   declare nb-c valant 10
   ;;; profondeur maximale acceptee pour un arbre d'etats
   declare prof valant 3
   ;;; colonne initiale du pion (place sur la ligne 0)
   declare c-ini valant 5
   ;;; probabilite d'avoir un 1 dans la grille : 1/proba
   declare proba-1 valant 3
affecter la grille de *obj-sys avec $(creer-grille nb-l nb-c proba-1)
affecter la dim-l de *obj-sys avec nb-l
affecter la dim-c de *obj-sys avec nb-c
affecter la profondeur de *obj-sys avec 0
affecter la prof-max  de *obj-sys avec prof
affecter le compteur-ses de *obj-sys avec 1
affecter la l-pos de *pion avec $(list (list 0 c-ini))
affecter la l-mouv de *pion avec $()
```

```
action $(progn (let ((new-se (GAMES-creer-se 'Deplace-pion
                                    (concat "Deplace-pion1")
                                    "Descend-KB"))
                     (old-se $S.Manager$))
                    (ecrit "** Nouvel exemple **" 'chemins-solutions)
                    (GAMES-va-dans-se new-se)
                    (send 'GAMES-copier-objet new-se old-se *pion)
                    (send 'GAMES-copier-objet new-se old-se *obj-sys)
                    (GAMES-va-dans-se old-se)
                    ;;; nom du nouveau se dans le fichier resultat
                    (ecrit (catenate "** S.Manager : " name^new-se)
                           'chemins-solutions)
                    ;;; position du pion dans la racine du nouveau se
                    (ecrit (catenate " Etat initial : "
                                     (explode l-pos^*pion))
                           'chemins-solutions)
                    (send 'GAMES-lancer new-se 'descend)
                    (S-pop-task)))
end-rule
```

3.5 Descend.ll

```
;;; initialisation de l'agenda et de la strategie d'un nouveau se
(S-set-agenda self^agenda-descend)
(S-set-strategy self^strategy-descend)
```

3.6 Descend.met

```
;;; ligne et colonne du pion d'apres la derniere position inscrite
;;; dans la liste des positions occupees par le pion

(defSmethod {Pion}:l (pion) () (caar l-pos^pion))

(defSmethod {Pion}:c (pion) () (cadar l-pos^pion))
```

3.7 Descend.rul

```
defrule prof-max
author steph
explanation
   si on a atteint la profondeur maximale fixee pour l'arbre d'etat,
   lancer un nouveau se
end-explanation
let *obj-sys un Objet-systeme
    et *pion un Pion
if profondeur^*obj-sys = prof-max^*obj-sys
then
action $(progn ;;; on augmente de 1 les compteurs des objets systemes
               ;;; de tous les ses, sinon comme les objets systemes
               ;;; sont differents dans chaque systeme on risque de
               ;;; creer des systemes avec le meme nom
               ;;; compteur-ses etant un champ insensible au contexte
               ;;; compteur-ses a la meme valeur dans tous les etats
               ;;; d'un systeme
               (mapc (lambda (se)
                             (GAMES-dans-se se
                                (let ((os (car (S-all-objects
                                                 'Objet-systeme))))
                                     (send 'compteur-ses
                                           os
                                           (1+ compteur-ses^os)))))
                     $liste-SEs$)
               ;;; creation et initialisation d'un nouveau se
               ;;; Deplace-pion
               (let ((new-se (GAMES-creer-se 'Deplace-pion
                                     (concat "Deplace-pion"
                                             compteur-ses^*obj-sys)
                                     "Descend-KB"))
                     (old-se $S.Manager$))
                    (GAMES-va-dans-se new-se)
                    (send 'GAMES-copier-objet new-se old-se *pion)
                    (send 'GAMES-copier-objet new-se old-se *obj-sys)
                    (send 'profondeur
                          (car (S-all-objects 'Objet-systeme)) 0)
                    (GAMES-va-dans-se old-se)
                    ;;; fichier resultats
                    (ecrit (catenate "** S.Manager : " name^new-se)
                           'chemins-solutions)
                    (ecrit (catenate " Etat initial : "
                                     (explode l-pos^*pion))
                           'chemins-solutions)
```

```
                    ;;; demarrage du raisonnement du nouveau se
                    (send 'GAMES-lancer new-se 'descend)
                    ;;; fermeture de l'etat de l'ancien se arrive
                    ;;; a prof-max
                    (S-set-singularity (S-current-state) 'Prof-max)))
end-rule

defrule alternatives
author steph
explanation
   envisage les differents deplacements possibles
   pour le pion parmi (-1, 0, +1)
end-explanation
let *pion un Pion
    et *obj-sys un Objet-systeme
    et *mouv un quelconque parmi $'(-1 0 1)
if $(if ;;; on regarde si on envisage des cases de la grille
        (and (>= (+ c^*pion *mouv) 0)
             (<= (+ c^*pion *mouv) (1- dim-c^*obj-sys))
             (<= l^*pion (1- dim-l^*obj-sys)))
        ;;; valeur de la case envisagee dans la grille
        (eq (aref grille^*obj-sys (1+ l^*pion) (+ c^*pion *mouv)) 0)
        ())
then
affecter la l-pos de *pion
            avec $(cons (list (1+ l^*pion) (+ c^*pion *mouv))
                        l-pos^*pion)
affecter la l-mouv de *pion avec $(cons *mouv l-mouv^*pion)
action $(S-pop-task)
end-rule

defrule profondeur
author steph
explanation
   la profondeur de l'arbre d'etats augmente de 1 apres
   le declenchement de la regle alternatives
end-explanation
let *objet-systeme un Objet-systeme
then
affecter la profondeur de *objet-systeme avec
   $(1+ profondeur^*objet-systeme)
;;; pour ne pas boucler sur la tache contenant la regle profondeur
action $(S-pop-task)
end-rule
```

```
defrule solution
author steph
explanation
   si le pion ne peut plus descendre dans la grille
   alors donner la solution
end-explanation
let *pion un Pion
    et *obj-sys un Objet-systeme
if $(cond ;;; on regarde les cases voisines de la grille
          ;;; attention aux bords de la grille
          ((eq l^*pion (1- dim-l^*obj-sys)) t)
          ((eq c^*pion 0)
           (and (eq (aref grille^*obj-sys
                          (1+ l^*pion) c^*pion) 1)
                (eq (aref grille^*obj-sys
                          (1+ l^*pion) (1+ c^*pion)) 1)))
          ((eq c^*pion (1- dim-c^*obj-sys))
           (and (eq (aref grille^*obj-sys
                          (1+ l^*pion) (1- c^*pion)) 1)
                (eq (aref grille^*obj-sys
                          (1+ l^*pion) c^*pion) 1)))
          (t (and (eq (aref grille^*obj-sys
                            (1+ l^*pion) (1- c^*pion)) 1)
                  (eq (aref grille^*obj-sys
                            (1+ l^*pion) c^*pion) 1)
                  (eq (aref grille^*obj-sys
                            (1+ l^*pion) (1+ c^*pion)) 1))))
then
action $(progn (print l-pos^*pion)
               ;;; imprime dans le fichier resultat le parcours
               ;;; du pion
               ;;; indique s'il a atteint le fond de la grille ou
               ;;; s'il est bloque par des 1
               (if (eq (caar l-pos^*pion) (1- dim-l^*obj-sys))
                   (ecrit l-pos^*pion 'chemins-solutions)
                   (ecrit (catenate " Pion bloque : "
                                    (explode l-pos^*pion))
                          'chemins-solutions))
               (S-set-singularity (S-current-state) 'Bloque))
end-rule
```

4 RÉSULTATS

4.1 Un exemple de grille

Comme exemple, nous avons généré une grille 10x10 (figure 4.1). Remarquons sur cet exemple, que de toute case 0, on peut atteindre le fond de la grille, excepté depuis le point `(4,5)` qui ne permet pas à un pion de descendre davantage. Tous les chemins aboutissant en `(4,5)` sont marqués "`Pion bloque : ...`" dans le fichier résultat. Les conditions initiales sont définies dans la règle `Initialise.rul`.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	1
2	0	1	0	0	1	0	0	1	0	1
3	1	0	1	0	1	0	0	0	0	1
4	0	0	0	0	1	0	0	0	0	1
5	0	1	0	0	1	1	1	0	1	1
6	1	0	0	1	1	1	0	0	1	0
7	0	0	0	1	0	1	0	0	0	0
8	0	1	0	1	0	0	1	0	1	0
9	1	0	1	0	0	0	1	1	0	0

Figure 4.1 : Exemple de grille 10x10

4.2 Liste des chemins

Chaque création d'un nouveau système expert est précisé dans le fichier résultat par "** S.Manager : ...". Dans cet exemple on utilise 62 systèmes à la fois. Dans cette application, on pourrait détruire les systèmes une fois leur arbre d'états développé entièrement à la profondeur maximum de prof-max. On écrit aussi dans le fichier résultat la position du pion dans la racine de l'arbre d'états d'un système, précédé de "Etat initial :". Les lignes où ne figurent que des listes de positions correspondent à une solution pour atteindre le fond de la grille depuis la position initiale (0,5) proposée.

```
** Nouvel exemple **
** S.Manager : Deplace-pion1
 Etat initial : ((0 5))
** S.Manager : Deplace-pion2
 Etat initial : ((3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion3
 Etat initial : ((6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 0) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 1) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion4
 Etat initial : ((6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 2) (5 2) (4 2) (3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion5
 Etat initial : ((6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 2) (5 3) (4 2) (3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion6
 Etat initial : ((6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 0) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 1) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
```

```
** S.Manager : Deplace-pion7
 Etat initial : ((6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 2) (5 2) (4 3) (3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion8
 Etat initial : ((6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 0) (7 1) (6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 1) (6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 1) (6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 1) (8 2) (7 2) (6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
((9 3) (8 2) (7 2) (6 2) (5 3) (4 3) (3 3) (2 3) (1 4) (0 5))
** S.Manager : Deplace-pion9
 Etat initial : ((3 5) (2 5) (1 4) (0 5))
 Pion bloque : ((4 5) (3 5) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion10
 Etat initial : ((6 6) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 6) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion11
 Etat initial : ((6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 8) (8 9) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
((9 9) (8 9) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion12
 Etat initial : ((3 6) (2 5) (1 4) (0 5))
 Pion bloque : ((4 5) (3 6) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion13
 Etat initial : ((6 6) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 6) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 6) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 6) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 6) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
```

```
** S.Manager : Deplace-pion14
 Etat initial : ((6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 8) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 9) (7 8) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
((9 9) (8 9) (7 8) (6 7) (5 7) (4 6) (3 6) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion15
 Etat initial : ((6 6) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 6) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 6) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 6) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 6) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion16
 Etat initial : ((6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 4) (8 5) (7 6) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 5) (8 5) (7 6) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 6) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 7) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 7) (7 8) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 8) (8 9) (7 8) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
((9 9) (8 9) (7 8) (6 7) (5 7) (4 7) (3 6) (2 5) (1 4) (0 5))
** S.Manager : Deplace-pion17
 Etat initial : ((3 5) (2 5) (1 5) (0 5))
 Pion bloque : ((4 5) (3 5) (2 5) (1 5) (0 5))
** S.Manager : Deplace-pion18
 Etat initial : ((6 6) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 4) (8 5) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 5) (8 5) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 7) (7 6) (6 6) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 7) (7 7) (6 6) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
** S.Manager : Deplace-pion19
 Etat initial : ((6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 4) (8 5) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 5) (8 5) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 7) (7 6) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 7) (7 7) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 7) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 8) (8 9) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
((9 9) (8 9) (7 8) (6 7) (5 7) (4 6) (3 5) (2 5) (1 5) (0 5))
** S.Manager : Deplace-pion20
 Etat initial : ((3 6) (2 5) (1 5) (0 5))
 Pion bloque : ((4 5) (3 6) (2 5) (1 5) (0 5))
** S.Manager : Deplace-pion21

...
```

4.3 Quelques arbres d'états

Comme on n'a pas détruit de systèmes experts au cours de la résolution, après avoir trouvé tous les chemins solution, on peut aller dans chaque système pour inspecter séparemment les arbres d'états construits. Les figures 4.2 et 4.3 donnent deux exemples de sous-arbres développés. On trouve des feuilles avec des singularités `prof-max`, lorsque le raisonnement est poursuivi dans un autre système et des singularités `bloque` lorsqu'un pion ne peut plus progresser dans la grille, qu'il soit bloqué sur la dernière ligne de la grille ou au milieu de la grille.

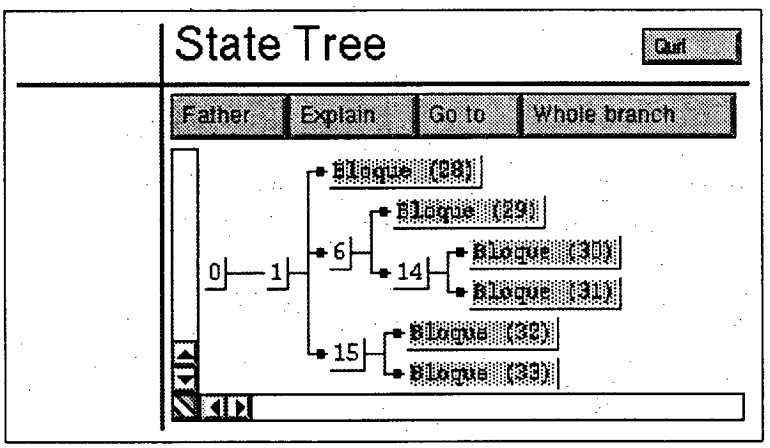

Figure 4.2 : Arbre d'états du système expert `Deplace-pion6`

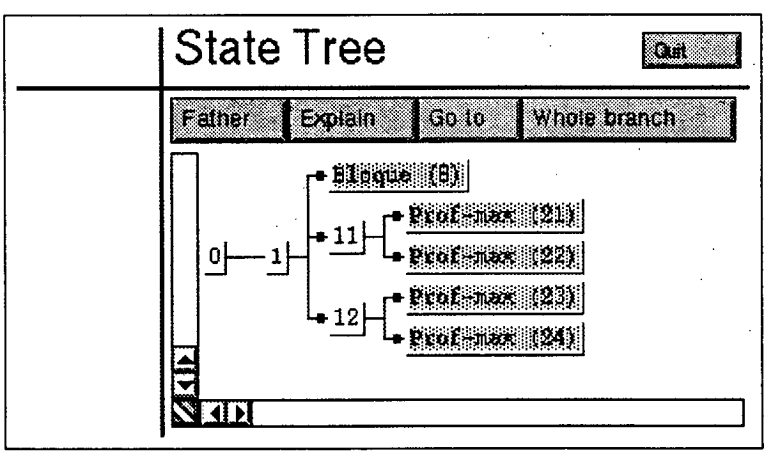

Figure 4.3 : Arbre d'états du système expert `Deplace-pion12`

Les états 28, 29 , 30 et 31 de la figure 4.2 correspondent à des chemins aboutissant en (9,1)... On retrouve ces résultats en allant dans l'état singulier ou en relisant le fichier

de sorties.

De la même façon, on analyse les singularités de l'arbre d'états de la figure 4.3. En regardant dans le fichier résultat après "** S.Manager : Deplace-pion12", on voit que l'état 8 correpond à un pion bloqué dans la grille et que le raisonnement de l'état 21 s'est poursuivi dans le système Deplace-pion13 avant de donner quatres solutions. De même, on s'aperçoit que le raisonnement de l'état 22 a été poursuivi dans le système Deplace-pion14 pour donner huit solutions.

5 CONCLUSION

GAMES utilise des fonctions de SMECI non documentées et s'appuie aussi sur des structures cachées. GAMES permet le développement d'applications manipulant simultanément plusieurs arbres d'etats en SMECI version 1.65-1. GAMES ne fonctionne pas sans quelques modifications, dans des versions antérieures de SMECI ou en version 1.65-2.

Nous avons documenté les principales fonctions de GAMES. Mais on ne donne pas dans ce rapport d'exemples de hiérarchies de systèmes experts. On peut en effet facilement envisager des systèmes experts dédiés à certaines tâches avec des comportements particuliers, comme par exemple une méthode de retour différente de la méthode de retour standard des systèmes du type `S.Manager`. Ces systèmes dédiés peuvent être des systèmes experts analysant ou modifiant les arbres d'états d'autres systèmes...

Références

[1] Rose Dieng. Cooperating expert systems for analysis of an expert system. In *4th International Symposium on Methodologies for Intelligent Systems*, pages 52–59, North-Holland, Charlotte, NC, USA, October 1989.

[2] ILOG. *SMECI Version 1.65 : Le manuel de référence.* 2 avenue Galliéni, BP 85, 94253 Gentilly Cedex FRANCE, Mai 1990.

[3] B. Neveu, B. Trousse, and O. Corby. SMECI : an Expert System Shell that fits Engineering Design. In *3er Simposium Internacional de Inteligencia Artificial*, Monterrey, N.L., Mexique, October 1990.

[4] Patrice Poyet, Philippe De La Cruz, Thierry Miléo, and Jean-Noël Loiseau. Récentes études en matière de simulation tactiques intelligentes. In *9ième journées internationales d'Avignon, les systèmes experts et leurs applications, conférence spécialisée intelligence artificielle et défense*, pages 149, session 4, Avignon, France, Mai-juin 1989.

Index des fonctions

(send 'GAMES-classe *se*) [*méthode*] 7
(send 'GAMES-nom-defaut *se*) [*méthode*] 8
(send 'GAMES-bc-defaut *se*) [*méthode*] 8
(GAMES-creer-se *classe nom-se . bc*) [*fonction*] 8
(send 'GAMES-av-cree *se*) [*méthode*] 9
(send 'GAMES-si-cree *se*) [*méthode*] 9
(send 'GAMES-retour *vieux-se nouveau-se*) [*méthode*] 10
(GAMES-trouver-se *nom*) [*fonction*] 10
(GAMES-trouver-se-classe *classe . nom*) [*fonction*] 10
(GAMES-trouver-tous-ses *classe*) [*fonction*] 10
(GAMES-S-find-collector *nom-fic*) [*fonction*] 11
(GAMES-trouver-se-regles *nom-fic*) [*fonction*] 11
(GAMES-charger-bc *bc*) [*fonction*] 11
(GAMES-charger-categs *fich*) [*fonction*] 12
(GAMES-charger-regles *fich*) [*fonction*] 12
(GAMES-charger-methodes *fich*) [*fonction*] 13
(GAMES-charger-objets *fich*) [*fonction*] 13
(GAMES-charger-ficlisp *fich*) [*fonction*] 13
(GAMES-detruire *se*) [*fonction*] 14
(send 'GAMES-va *se*) [*méthode*] 14
(GAMES-va-dans-se *se*) [*fonction*] 14
(GAMES-dans-se *se . expr*) [*macro*] 14
(send 'GAMES-lancer *se . tache*) [*méthode*] 15
(send 'GAMES-av-lancer *se*) [*méthode*] 15
(send 'GAMES-ap-lancer *se*) [*méthode*] 16
(GAMES-copier-objet *se1 se2 obj*) [*méthode*] 16
(GAMES-cree-editeur *x y*) [*fonction*] 16
(GAMES-raff-editeur *edit*) [*fonction*] 17

Unité de Recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers Lès Nancy Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex

ISSN 0249 - 0803